Fast Data Encoding and Decoding Algorithms

Self Paper to Ph.D. Thesis

Jiří Walder

Department of Computer Science
VŠB – Technical University of Ostrava
Czech Republic
Abstract

Data compression has been widely applied in many data processing areas. Compression methods utilize variable-length codes with shorter codes assigned to symbols or groups of symbols that appear in the data frequently. Fibonacci, Elias-delta, and Elias-Fibonacci codes are representatives of these codes, and are often utilized for the compression of numbers. The time consumption of encoding as well as decoding algorithms is important for some applications in the data processing area. In this case, the efficiency of these algorithms is extremely important. Some works related to fast decoding of variable-length codes have been published in recent years. Fast encoding algorithms for these codes have not yet been studied. In this work, we introduce fast encoding and decoding algorithms. Our fast encoding algorithms are up to 12.5× faster than conventional bit-oriented algorithms and our decoding algorithms are up-to 8.9× faster than the conventional decoding algorithms.

Keywords: compression, variable-length code, fast encoding algorithm, Elias-delta code, Fibonacci codes of order 2 and 3, Elias-Fibonacci code
Contents

1 Introduction 1

2 Variable-Length Codes 2
  2.1 Elias-delta code .................................................. 2
  2.2 Fibonacci Codes .................................................. 2
    2.2.1 Fibonacci Code of Order 2 .................................. 2
    2.2.2 Fibonacci Codes of Higher Orders ......................... 3
  2.3 Elias-Fibonacci Code ............................................ 4

3 Preliminaries of Encoding and Decoding Algorithms 4

4 Fibonacci Shift Operations and their Computation 5
  4.1 Fibonacci Shift Operations ..................................... 5
  4.2 Efficient Computation of Fibonacci Right Shift ............. 5
  4.3 Efficient Computation of Fibonacci Left Shift ............. 7

5 Efficient Computation of Nearest Lower Fibonacci Sum 8

6 Fast Encoding Algorithms 10
  6.1 Fast Elias-delta and Elias-Fibonacci Encoding Algorithms .... 10
  6.2 Fast Encoding of Fibonacci Code of Order 2 .................. 12
  6.3 Fast Encoding of Fibonacci Code of Order 3 .................. 15

7 Fast Decoding Algorithms 17
  7.1 Fast Elias-delta and Elias-Fibonacci Decoding Algorithms ... 17
  7.2 Fast Fibonacci Decoding Algorithm ............................ 21

8 Experimental Results 24
  8.1 Compression Ratio ............................................... 25
  8.2 Encoding Algorithms ............................................. 25
  8.3 Decoding Algorithms ............................................. 26

9 Conclusion 27

Author’s Bibliography 28

Bibliography 29
1. Introduction

Data compression has been widely applied in many data processing areas. Various compression algorithms have been developed for processing text documents, images, video, etc. In particular, data compression is of the foremost importance and has been well researched as it has been presented in excellent books [26, 35].

In contrast with fixed-length codes, statistical methods utilize variable-length codes [27], with the shorter codes assigned to symbols or groups of symbols that have a higher probability of occurrence. People who design and implement variable-length codes have to deal with these two problems: (1) assigning codes that are uniquely decodable and (2) assigning codes with the minimum average size. Several variable-length codes such as Elias [12], Fibonacci [13, 9], Elias-Fibonacci [4], Golomb [16, 34], and Huffman codes [17] are well known representatives of variable-length codes.

Fast encoding/decoding algorithms have been intensively studied in a connection with video and audio encoders/decoders. Moreover, these studies often integrate the variable-length codes in the encoders and decoders. The result is the following recommendations: T.81 for a JPEG encoder [19] and H.261 for an MPEG encoder [18]. These encoders utilize Exponential Golomb [31] or Huffman codes. Many works and patents (e.g. [14, 20, 30]) deal with fast encoders/decoders for these codes. In this case, a codeword is predefined for each coded number; the size of the encoding/decoding table then corresponds to the size of the number domain used. Therefore, these algorithms can be used only for small domains of coded numbers, e.g. 8 or 16 bits; they are not useful for general universal codes where the domain can be large (e.g. 32 bit-length numbers). Since these domains are often used in the data processing area or text file compression [34, 29], other algorithms utilizing more general variable-length codes are investigated. Compression properties of the variable-length codes have been compared in several works [32, 3]. In these articles, Fibonacci, Elias-delta, and Elias-Fibonacci codes overcome other variable-length codes.

The time consumption of compression and decompression is sometimes critical; therefore, efficient decompression algorithms have been studied in many works related to decompression of data structures [28, 15, 11, 2] or text files [25, 10].

The variable-length codes for larger domains are not often utilized since efficient algorithms for encoding and decoding have been not known. There are only very few works dealing with fast decoding algorithms for Fibonacci codes [21, 22]. In this paper, we propose fast encoding and decoding algorithms for Elias-delta, Fibonacci of order 2 and 3, and Elias-Fibonacci codes and we compare the efficiency of these algorithms with conventional algorithms.

In Section 2, we depict an introduction to variable-length codes and the new Elias-Fibonacci code is introduced. We briefly describe the main ideas of conventional and fast algorithms in Section 3 and we introduce a theoretical background to Fibonacci codes necessary for fast Fibonacci encoding and decoding algorithms in Sections 4 and 5. We introduce fast encoding algorithms in Section 6. Basic principles of fast decoding algorithms and the fast decoding algorithms are described in Section 7. In Section 8, we compare conventional and fast algorithms and we also show some compression properties of these codes. We conclude this paper in Section 9.
2 Variable-Length Codes

In this section, we give an overview of variable-length codes used in this paper. In Sections 2.1, we start with the well-known Elias-delta code. In Section 2.2, the Fibonacci family codes are described. Our new Elias-Fibonacci code, introduced in [4], is put forward in Section 2.3.

2.1 Elias-delta code

Elias-delta code was introduced by Peter Elias [12]. In the Elias-delta code, a positive integer \( n \) is encoded as:

\[
E_\delta(n) = 0_{Z-1} B(L) B'(n)
\]

where \( B'(n) \) without the highest 1-bit, \( L \) is the bit-length of the standard binary representation \( B(n) \), \( B(L) \) is the standard binary representation of \( L \) and \( Z \) is the bit-length of \( B(L) \).

2.2 Fibonacci Codes

Fibonacci codes are closely related to the Fibonacci representation of an integer and are based on Fibonacci numbers [24]. A Fibonacci number \( F_i^{(m)} \) of order \( m \) is recursively defined as follows:

\[
F_i^{(m)} = F_{i-1}^{(m)} + F_{i-2}^{(m)} + \ldots + F_{i-m}^{(m)}, \quad \text{for } i \geq 1,
\]

where \( F_{-m+1}^{(m)} = F_{-m+2}^{(m)} = \ldots = F_{-2}^{(m)} = 0 \)

and \( F_{-1}^{(m)} = F_{0}^{(m)} = 1 \)

Definition 1 (The Fibonacci representation of order \( m \) for an integer \( n \))

A number \( F_i^{(m)}(n) = a_p a_{p-1} \ldots a_0, a_i \in \{0, 1\}, 0 \leq i \leq p, \) is the Fibonacci representation of a positive integer \( n \) iff:

\[
\sum_{i=0}^{p} a_i F_i^{(m)} = n
\]

and there is no run of \( m \) consecutive 1-bits in \( F_i^{(m)}(n) \).

Since this representation has the property of not containing any sequence of \( m \) consecutive 1-bits [9], we can construct Fibonacci codewords; each codeword includes a sequence of \( m \) consecutive 1-bits in the lowest bits as a delimiter.

2.2.1 Fibonacci Code of Order 2

The reverse Fibonacci code of order 2\(^1\) has been introduced in [13]. Since fast decoding algorithms [4, 23] have to read bits of codewords from the lowest to the highest bit, we must utilize

\(^1\)This code is commonly known as the Fibonacci code; however, we use the prefix reverse since it corresponds to the fact that the Fibonacci representation of this code is reversed. Consequently, we need to distinguish this code and the generalized Fibonacci codes of order \( \geq 2 \) introduced by Apostolico and Fraenkel in [9] not using the reverse Fibonacci representation (see Section 2.2.2).
the reverse Fibonacci representation. In the reverse Fibonacci representation, we reverse the bits so that the highest 1-bit becomes the lowest 1-bit and so on; when a bit of the reverse Fibonacci representation is read, we always know the order $i$ of the $F_i$ number in the representation. The reverse Fibonacci code of order 2, denoted by $\mathcal{F}^{(2)}(n)$, is obtained by appending an additional 1-bit as the lowest bit to the reverse Fibonacci representation, i.e.:

$$\mathcal{F}^{(2)}(n) = reverse(F(n)^{(2)})1$$

### 2.2.2 Fibonacci Codes of Higher Orders

In the case of Fibonacci codes of order 3 and higher, it is not possible to create codewords by only appending $1_{m-1}$ bits to the reverse Fibonacci representation $F^{(m)}(n)$ as in the case of the reverse Fibonacci code of order 2. In [9], authors introduced the generalized Fibonacci code of order $m$ ($m \geq 2$). The authors also depicted a proof of the completeness of the code. Each codeword of this code includes a sequence $1_m$ in the lowest bits as the delimiter. The Fibonacci sum $S_{g}^{(m)}$ (See [9] or Ph.D. Thesis for definition) must be used to construct a valid prefix code.

Consequently, the generalized Fibonacci code $\mathcal{F}_{C}^{(m)}(n)$ was originally defined by Apostolico and Fraenkel in [9]. Like in the case of the reverse Fibonacci code of order 2, we need a reverse variant of these codes appropriate for fast decoding algorithms. The reverse variant has been introduced by Shmuel T. Klein at al. in [22] and we use it in our fast encoding and decoding algorithms described later. This modification is denoted by $\mathcal{F}^{(m)}(n)$ in the following text. The reverse Fibonacci code $\mathcal{F}^{(m)}(n)$ is defined as follows:

1. If $n = 1$ then $\mathcal{F}^{(m)}(n) = 1_m$. END.
2. If $n = 2$ then $\mathcal{F}^{(m)}(n) = 01_m$. END.
3. Find $g$ such that $S_{g-2}^{(m)} < n \leq S_{g-1}^{(m)}$.
4. Compute $Q = n - S_{g-2}^{(m)} - 1$.
5. Compute $F^{(m)}(Q)$.
6. Let $L(F^{(m)}(Q))$ be the bit-length of $F^{(m)}(Q)$.
7. Compute $z = g - 1 - L(F^{(m)}(Q))$.
8. The reverse Fibonacci code of order $m$ with bit-length $m + g$ is then the concatenation:

$$\mathcal{F}^{(m)}(n) = reverse(F^{(m)}(Q))0_z01_m$$
2.3 Elias-Fibonacci Code

In this section, we describe the Elias-Fibonacci code: our new code introduced in [4]. It consists of two parts. The second part is $B'(n)$, the binary representation of a number $n$ without the highest 1-bit. The first part is $L(n)$ (or $L$), the bit-length of $B(n)$, encoded with the Fibonacci code of order 2 (i.e. $\mathcal{F}^{(2)}(L)$). In other words, if we reach two 1-bits in a codeword, we read $L$ and we know that we must read $L - 1$ bits to complete $B'(n)$. The codeword of the Elias-Fibonacci code is then the concatenation:

$$EF(n) = \mathcal{F}^{(2)}(L)B'(n)$$

3 Preliminaries of Encoding and Decoding Algorithms

Conventional encoding and decoding algorithms, described in Ph.D. Thesis, process data bit-by-bit which is rather time consuming. The main idea of fast encoding and decoding algorithms, is to encode and decode data in a bigger chunk of bits called segment. In Figure 1, we see a general schema of conventional and fast encoding and decoding algorithms. The fast algorithms described in 6 then encode or decode codewords segment-by-segment instead of bit-by-bit. We see the segment is used to handle codewords – it is utilized in the output of encoding and in the input of decoding. The segments usually have the bit-length of 8, 16, 32 or more bits to fit in a processor register.

Figure 1: A general schema of conventional and fast encoding and decoding algorithms for the 8bit segment
Our fast decoding algorithms are based on a finite automaton [33]. The state of the automaton depends on a value of the current segment read from an input stream and the previous automaton state. For each automaton state, a precomputed mapping table defines the output decoded numbers and the new automaton state.

For both conventional and fast encoding algorithms, we assume a two-step process encoding a set of integers into an output stream. In the first step, we encode a number in a temporary buffer. In the second step, this buffer is written in the output stream.

4 Fibonacci Shift Operations and their Computation

4.1 Fibonacci Shift Operations

The Fibonacci shift operations, introduced in [4, 23, 6], are required for the bit manipulation in the fast encoding and decoding algorithms of the Fibonacci code described in Sections 6–7.

Definition 2 (The Fibonacci shift operations)

Let \( F(n) = a_0 a_1 a_2 \ldots a_p \) be the Fibonacci representation of an integer \( n \) and \( k \geq 0 \) be an integer. The \( k \)-th Fibonacci left shift \( n \ll_F k \) is defined as follows

\[
 n \ll_F k = a_0 a_1 a_2 \ldots a_p \underbrace{00 \ldots 0}_{k \text{ bits}}
\]

and \( k \)-th Fibonacci right shift is defined as follows

\[
 n \gg_F k = a_0 a_1 a_2 a_k + 1 a_k
\]

and \( k \)-th Extended Fibonacci right shift is defined as follows

\[
 n \gg_{F_{ext}} k = a_0 a_1 a_2 a_k + 1 a_k \cdot a_{k-2} a_{k-1} \ldots a_0
\]

Fibonacci and Extended Fibonacci right shifts are not inverse operations to the Fibonacci left shift since the Fibonacci right shift truncates the bits of \( F(n) \) from the position \( k - 1 \) and the Extended Fibonacci right shift truncates the bits of \( F(n) \) from the position \( k - 2 \) (these bits belong to insignificant Fibonacci numbers of the Extended Fibonacci representation since \( F_{-1}^m = 0 \) for \( i < -1 \)).

4.2 Efficient Computation of Fibonacci Right Shift

In this section, we describe an efficient computation of the Fibonacci right shift introduced in [6]. The computation is utilized during a fast Fibonacci encoding algorithm to obtain individual segments of the bit-length \( S \) of the Fibonacci representation (see Sections 6.2 and 6.3).

Let us consider an interval \([n_{\min}, n_{\max}]\) of all numbers with the \( k \)-th Fibonacci right shift equal to \( x \), i.e., \( \forall n \in [n_{\min}, n_{\max}] : n \gg_F k = x \). A table including all intervals for a domain is called the Fibonacci right shift encoding interval table (FRSEIT). Each row of FRSEIT includes the following values (see Table 1 for a fragment of FRSEIT):

- \( k \) – the parameter of the Fibonacci right shift
• \(x\) – the result of the Fibonacci right shift
• \([n_{\text{min}}, n_{\text{max}}]\) – the interval of numbers having the result \(x\) of the Fibonacci right shift
• \(F(x)\) – the Fibonacci representation of the result \(x\)
• \(\text{enc}(F(x))\) – the encoded Fibonacci representation – the standard binary representation of \(F(x)\). This encoding is symbolically represented by changing an \(F\) subscript to a 2 subscript (see Table 1 for an example).
• \(L(F(x))\) – the bit-length of the Fibonacci representation \(F(x)\)

FRSEIT is then utilized to find the \(k\)-th Fibonacci right shift of \(n\); for the \(k\)-th Fibonacci right shift we find the row where \(n \in \langle n_{\text{min}}, n_{\text{max}}\rangle\). In this row, we can directly read the result \(x\) of the Fibonacci right shift \(n \gg_F k\) as well as the Fibonacci representation \(F(x)\). When we consider a binary search algorithm, this procedure has logarithmic complexity.

Table 1: A fragment of FRSEIT for the Fibonacci code of order 2 and the 8-th and 16-th Fibonacci right shifts

<table>
<thead>
<tr>
<th>Interval for (k = 8)</th>
<th>Interval for (k = 16)</th>
<th>(x)</th>
<th>(F(x))</th>
<th>(\text{enc}(F(x)))</th>
<th>(L(F(x)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n_{\text{min}})</td>
<td>(n_{\text{max}})</td>
<td>(n_{\text{min}})</td>
<td>(n_{\text{max}})</td>
<td>(0)</td>
<td>(0)</td>
</tr>
<tr>
<td>0</td>
<td>54</td>
<td>0</td>
<td>2,583</td>
<td>(0_F)</td>
<td>(0_{10})</td>
</tr>
<tr>
<td>55</td>
<td>88</td>
<td>2,584</td>
<td>4,180</td>
<td>1</td>
<td>(1_F)</td>
</tr>
<tr>
<td>89</td>
<td>143</td>
<td>4,181</td>
<td>6,764</td>
<td>2</td>
<td>(10_F)</td>
</tr>
<tr>
<td>144</td>
<td>198</td>
<td>6,765</td>
<td>9,348</td>
<td>3</td>
<td>(100_F)</td>
</tr>
<tr>
<td>199</td>
<td>232</td>
<td>9,349</td>
<td>10,945</td>
<td>4</td>
<td>(101_F)</td>
</tr>
<tr>
<td>233</td>
<td>287</td>
<td>10,946</td>
<td>13,529</td>
<td>5</td>
<td>(1000_F)</td>
</tr>
<tr>
<td>288</td>
<td>321</td>
<td>13,530</td>
<td>15,126</td>
<td>6</td>
<td>(1001_F)</td>
</tr>
<tr>
<td>322</td>
<td>376</td>
<td>15,127</td>
<td>17,710</td>
<td>7</td>
<td>(1010_F)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1,275</td>
<td>1,308</td>
<td>59,898</td>
<td>61,494</td>
<td>27</td>
<td>(1001001_F)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2,173</td>
<td>2,206</td>
<td>102,085</td>
<td>103,681</td>
<td>46</td>
<td>(10010101_F)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

To improve logarithmic time complexity of binary searching in FRSEIT, we must consider properties of Fibonacci numbers described in details in Ph.D. Thesis. Using these properties, we can estimate the result of the \(k\)-th Fibonacci right shift and the estimation is then utilized to search the correct result of the \(k\)-th Fibonacci right shift. The estimation is computed according to the following theorem.

**Theorem 1** (An estimation of the \(k\)-th Fibonacci right shift)

Let an estimation of the \(k\)-th Fibonacci right shift be:

\[
n \gg_F k \approx \left\| \frac{n}{\phi_m^k} \right\|
\]
where \( m \) is the Fibonacci order and \( k \) is the parameter of the Fibonacci right shift. The error of the estimation is:
\[
\varepsilon \in \{0, 1\}.
\]
The proof of the theorem is included in Ph.D. Thesis.

**Algorithm 1:** A computation of the \( k \)-th Fibonacci right shift of \( n \)

For the computation of the correct result of the \( k \)-th Fibonacci right shift \( n >>_F k = x \), we use Algorithm 1. In Line 1, we estimate the result using Theorem 1; we obtain the estimation \( x' = \left\| \frac{n}{\phi_m^k} \right\| \). The \( x' \) value is also the row order in FRSEIT. In Line 2, we compare the input number \( n \) (the number being Fibonacci shifted) with the interval in the row order \( x' \). If the number is not in the interval, we simply subtract the estimated result by the value 1 and receive the correct result \( x \) of the \( k \)-th Fibonacci right shift. This means that our estimation method decreases logarithmic complexity of the computation to two attempts at the most.

**Example 1** (An efficient computation of the Fibonacci right shift)

Let us consider \( n = 280 \) with the Fibonacci representation \( F(280) = 10010100000_F \). We calculate the 8-th Fibonacci right shift of \( n \), \( 280 >>_F 8 \), by the following procedure. First, we calculate the estimation of the 8-th Fibonacci right shift for this number by means of Theorem 1 as follows:
\[
280 >>_F 8 \approx \left\| \frac{280}{46.9787} \right\| = \|5.9601\| = 6
\]

After checking the interval in the 6th row of FRSEIT, we see the number \( 280 \notin [288; 321] \); therefore, the estimation is not correct and we must decrease the estimated value by 1. Consequently, the correct result of the 8-th Fibonacci right shift \( 280 >>_F 8 \) is 5. We can then read its Fibonacci representation \( F(5) = 1000_F \) in the 5th row of the table.

### 4.3 Efficient Computation of Fibonacci Left Shift

In [4], we introduce an efficient computation of the Fibonacci left shift based on Fibonacci numbers and the Extended Fibonacci right shift. It is used in the fast Fibonacci decoding algorithm described in Section 7.2.

**Theorem 2** (The computation of the \( k \)-Fibonacci left shift for order \( m = 2 \))

The \( k \)-Fibonacci left shift of a number \( n \) for order \( m = 2 \) is:
\[
n <<_F k = F_k^{(2)} \times n + F_{k-1}^{(2)} \times (n >>_{F_{ext}} 1)
\]
8 Fast Data Encoding and Decoding Algorithms

The proof of the theorem is included in Ph.D. Thesis

**Theorem 3** (The computation of the k-Fibonacci left shift for order \( m = 3 \))

The k-Fibonacci left shift of a number \( n \) for order \( m = 3 \) is:

\[
n \ll_F k = F_{k-1}^{(3)} \times n + (F_{k-2}^{(3)} + F_{k-3}^{(3)}) \times (n \gg_{F_{\text{ext}} 1}) + F_{k-2}^{(3)} \times (n \gg_{F_{\text{ext}} 2})
\]

The proof of the theorem is included in Ph.D. Thesis

The computation of the Fibonacci left shift according to Theorem 2 or 3 is used in the fast Fibonacci decoding algorithm described in Section 7.2. This computation requires the 1-st Extended Fibonacci right shift for Fibonacci of order 2 and the 2-nd Extended Fibonacci right shift for Fibonacci of order 3. Since this shift is applied only on individual segments and the number of integers in a segment is low, we utilize a precomputed table of the 1-st and 2-nd Extended Fibonacci right shifts for all integers in a segment.

**Example 2** (A computation of the Fibonacci left shift)

Let us consider \( n = 32 \) and its Fibonacci representation \( F^{(2)}(32) = 1010100^{(2)} \). The 1-st Extended Fibonacci right shift of 32 is: \( 32 \gg_{F_{\text{ext}} 1} = 101010 \). The 3-nd Fibonacci left shift is computed as follows by means of Theorem 2:

\[
n \ll_F 3 = F_{3-1}^{(2)} \times 32 + F_{3-2}^{(2)} \times 20 = 3 \times 32 + 2 \times 20 = 96 + 40 = 136
\]

5 Efficient Computation of Nearest Lower Fibonacci Sum

To encode the Fibonacci code of order 3 and higher by the conventional as well as fast algorithms (see Section 6.3), we need to find the nearest lower Fibonacci sum; it is used to determine the codeword bit-length. A simple method is to use a binary search algorithm with logarithmic complexity finding the Fibonacci sum \( S_g \) in a precomputed table.

In Ph.D. Thesis we have shown that \( S_g \) depends logarithmically on \( n \) value. Therefore, we estimate the \( g \) value by another logarithmic function which can be calculated without the usage of floating-point arithmetic; we use the bit-length of the standard binary representation of \( n \) which is the logarithmic function \( L = L(n) = \lceil \log_2(n + 1) \rceil \). Evidently, for the Fibonacci code of order 3, the value \( \phi_3 = 1.8392868 \) is close to the base of the logarithm (the value 2).

To obtain the correct value \( k \) from the estimated value, we define the \( \text{EstimateFibSum} \) table including the following values:

- \( L = L(n) \), the bit-length of the standard binary representation of \( n \)
- The \( g \) value of the interval \((S_{g-2}, S_{g-1}]\)
- The interval \((S_{g-2}, S_{g-1}]\). This interval is calculated for the lowest \( n \) with the bit-length \( L \).

The \( \text{EstimateFibSum} \) table, precomputed for 32 bit-length numbers, is shown in Table 2. In Algorithm 2, we see how to find the correct nearest lower Fibonacci sum using this table.
5. Efficient Computation of Nearest Lower Fibonacci Sum

**Algorithm 2: Finding the nearest lower Fibonacci sum**

```plaintext
input: n, a positive integer
output: g, the index of the nearest lower Fibonacci sum S_{g-2}
1 L ← L(n);
2 g ← EstimateFibSum[L];
3 while S_{g-1} < n do g ← g + 1;
```

Table 2: The EstimateFibSum table for 32 bit-length numbers

<table>
<thead>
<tr>
<th>L</th>
<th>g</th>
<th>S_{g-2}</th>
<th>S_{g-1}</th>
<th>L</th>
<th>g</th>
<th>S_{g-2}</th>
<th>S_{g-1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>11</td>
<td>11</td>
<td>600</td>
<td>1,104</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>12</td>
<td>13</td>
<td>2,031</td>
<td>3,736</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>13</td>
<td>14</td>
<td>3,736</td>
<td>6,872</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td>15</td>
<td>14</td>
<td>15</td>
<td>6872</td>
<td>12,640</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>15</td>
<td>28</td>
<td>15</td>
<td>16</td>
<td>12,640</td>
<td>23,249</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>28</td>
<td>52</td>
<td>16</td>
<td>17</td>
<td>23,249</td>
<td>42,762</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>52</td>
<td>96</td>
<td>17</td>
<td>18</td>
<td>42,762</td>
<td>78,652</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>96</td>
<td>177</td>
<td>18</td>
<td>19</td>
<td>78,652</td>
<td>144,664</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>177</td>
<td>326</td>
<td>19</td>
<td>20</td>
<td>144,664</td>
<td>266079</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>326</td>
<td>600</td>
<td>20</td>
<td>22</td>
<td>489,396</td>
<td>900,140</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L</th>
<th>g</th>
<th>S_{g-2}</th>
<th>S_{g-1}</th>
<th>L</th>
<th>g</th>
<th>S_{g-2}</th>
<th>S_{g-1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>23</td>
<td>900,140</td>
<td>1,655,616</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>24</td>
<td>1,655,616</td>
<td>3,045,153</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>25</td>
<td>3,045,153</td>
<td>5,600,910</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>26</td>
<td>5,600,910</td>
<td>10,301,680</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>27</td>
<td>10,301,680</td>
<td>18,947,744</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>28</td>
<td>18,947,744</td>
<td>34,850,335</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>30</td>
<td>64,099,760</td>
<td>117,897,840</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>31</td>
<td>117,897,840</td>
<td>216,847,936</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>32</td>
<td>216,847,936</td>
<td>398,845,537</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>33</td>
<td>398,845,537</td>
<td>733,591,314</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>34</td>
<td>733,591,314</td>
<td>1,349,284,788</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>35</td>
<td>1,349,284,788</td>
<td>2,481,721,640</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 also shows the maximal error of the g value estimation. The estimation error is not greater than the difference of the g value between two consecutive rows in the table. Evidently, the maximal number for a specific bit-length L can belong to the interval specified in the next row of the table. Therefore, the maximal difference of g between two consecutive rows of the EstimateFibSum table is 2; see rows L ∈ \{11, 12\}, L ∈ \{19, 20\}, and L ∈ \{26, 27\}. As result, the precise g value is found by maximally two loops of the cycle in the algorithm. Since, a value can be maximally found in three Fibonacci sum intervals, i.e. two comparisons must be processed at the most, this estimation provides lower time complexity than the binary search algorithm. Moreover, we do not utilize floating-point arithmetic.

**Example 3 (A computation of the nearest lower Fibonacci sum)**

Let us consider n = 2,048 = 1000 0000 0000_2. The bit-length of the standard binary representation L = L(n) = 12. In the 12th row of EstimateFibSum, we read the estimated value g = 13 and the interval S_{g-2} = S_{13-2} = S_{11} = 2,031, S_{g-1} = S_{13-1} = S_{12} = 3,736. Since S_{11} ≤ n ≤ S_{12}, it is a correct interval; the nearest lower Fibonacci sum is S_{11} and g = 13.

Let us consider n = 511 = 1 1111 1111_2. The bit-length of the standard binary representation is L = L(n) = 9. In the 9th row of EstimateFibSum, we read the estimated value g = 9 and the interval S_{g-2} = S_{9-2} = S_{7} = 177, S_{g-1} = S_{9-1} = S_{8} = 326. Since S_{8} < n, it is not a correct interval. We must check the next interval with g = 9 + 1 = 10, and bounds S_{g-2} = S_{10-2} = S_{8} = 326, S_{g-1} = S_{10-1} = S_{9} = 600. Since S_{8} ≤ n ≤ S_{9}, the correct nearest lower Fibonacci sum is S_{8} and g = 10.
6 Fast Encoding Algorithms

Whereas the conventional encoding algorithms process data bit by bit, the fast algorithms process data by a higher number of bits. Since the Elias-delta and Elias-Fibonacci encoding algorithms are simpler, they encode numbers only using precomputed tables (see Section 6.1). Fast Fibonacci encoding algorithms described in Sections 6.2 and 6.3 utilize the efficient computation of the Fibonacci right shift described in Section 4.2 to compute individual segments of the Fibonacci representation in a codeword. In the case of Fibonacci of order 3, we also utilize the efficient computation of the nearest lower Fibonacci sum described in Section 5.

6.1 Fast Elias-delta and Elias-Fibonacci Encoding Algorithms

In fast Elias-delta and Elias-Fibonacci encoding algorithms, we separately encode both parts of the Elias-delta code $E_\delta(n) = 0_{Z-1}B(L)B'(n)$ as well as the Elias-Fibonacci code $EF(n) = \text{reverse}(F^{(2)}(L))B(n)$ (see Sections 2.1 and 2.3): the first part of codewords is a prefix, $0_{Z-1}B(L)$ or $\text{reverse}(F^{(2)}(L))$, and the second part is the binary part, $B(n)$ – the standard binary representation of $n$ or $B'(n)$ – the standard binary representation of $n$ without the highest 1-bit. The fast Elias-delta and Elias-Fibonacci encoding algorithms are shown in Algorithms 3 and 4, respectively. Although these algorithms encode a coded number in more bits than the conventional algorithms, they do not use the concept of $S$ bit-length segments like fast Fibonacci encoding algorithms described in next sections. Since encoding of $B(n)$ or $B'(n)$ is rather straightforward, we must aim our effort to encoding the bit-length in the prefix. Due to the fact that the count of various bit-lengths is relatively low, we can create short tables to encode the bit-length to the prefix of a codeword.

In more detail, the prefixes of Elias-delta and Elias-Fibonacci codes are directly encoded by means of tables called the $EDTAB$ and $EFTAB$ tables, respectively. These tables include the encoded prefixes for each bit-length $L(n)$ in the $PREFIX$ column. The bit-lengths of these prefixes are stored in the $LEN$ column. As mentioned previously, these tables are short; they include only 16 rows for all 16 bit-length coded numbers (see Table 3) and 32 rows for all 32 bit-length coded numbers. In Lines 2 and 3 of Algorithms 3 and 4, these tables are utilized, the prefix is then written in Line 4.

After encoding the prefix, $B(n)$ or $B'(n)$ must be written in $buffer$. In the case of Elias-Fibonacci, we do not need any encoding of the standard binary representation; $B(n)$ is directly written in $buffer$. However, in the case of Elias-delta, we must compute $B'(n)$. We know the bit-length $L(n)$ and we compute $B'(n)$ by means of the following function:

$$B'(n) = n \text{ AND } (1 << (L(n) - 1))^*$$

(6.1)

where $*$ denotes the 1’s complement of the standard binary representation and AND is a binary operator. We can save some execution time by preparing the $COMP$ column of $EDTAB$ including all $(1 << (L(n) - 1))^*$ (see Table 3). Consequently, $B'(n)$ is computed as follows:

$$B'(n) = n \text{ AND } EDTAB[L(n)].COMP$$
Table 3: EDTOB and EFTOB tables for all Elias-delta and Elias-Fibonacci codewords of 16 bit-length coded numbers

<table>
<thead>
<tr>
<th>$L = L(n)$</th>
<th>EDTOB</th>
<th>EFTOB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PREFIX</td>
<td>LEN</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1111111111111111</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>00100</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>00101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>00110</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>00111</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>000100</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>000101</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>0001010</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>0001101</td>
<td>7</td>
</tr>
<tr>
<td>12</td>
<td>0001110</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>0001111</td>
<td>7</td>
</tr>
<tr>
<td>14</td>
<td>00011011</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>00011111</td>
<td>7</td>
</tr>
<tr>
<td>16</td>
<td>000010000</td>
<td>9</td>
</tr>
</tbody>
</table>

Algorithm 3: A fast encoding algorithm for the Elias-delta code

This computation is utilized in Line 5 of Algorithm 3. In the last two lines of both algorithms, $B(n)$ or $B'(n)$ is written in buffer and the bit-length is computed.

Algorithm 4: A fast encoding algorithm for the Elias-Fibonacci code

Example 4 (The fast Elias-delta encoding algorithm)
Let us have a number \( n = 58 \) and its standard binary representation \( 111010_2 \). In the first line of Algorithm 3, we compute \( L = L(58) = 6 \). In Lines 2–3, we obtain prefixLen=5 and prefix=00110 using EDTAB. The prefix is stored in buffer starting the position \( 6 - 1 = 5 \): buffer=00110XXXXX (see Line 4). In Line 5, we compute \( 58 \text{ AND } 1111111111011111_2 = 11010_2 = 26 \). This value is stored in buffer in Line 6: buffer=0011011010. The bit-length of the codeword is computed in the last line: \( \text{len} = 6 + 5 - 1 = 10 \).

### 6.2 Fast Encoding of Fibonacci Code of Order 2

A fast encoding algorithm for the Fibonacci of code order 2 is shown in Algorithm 5. This algorithm utilizes an efficient computation of the Fibonacci right shift by means of \( \text{FRSEIT} \) described in Section 4.2. Two indices of \( \text{FRSEIT}[k][x] \) address the \( x \)-th row of \( \text{FRSEIT} \) and the \( k \)-th Fibonacci right shift.

```plaintext
input : \( n \), a positive integer
output: buffer, a Fibonacci of order 2 codeword of \( n \) with the bit-length \( \text{len} \)

1. \( k \leftarrow S \);
2. if \( n < F_k \) then
   3. \( \text{len} \leftarrow \text{FRSEIT}[k][n].LFx; \)
   4. \( \text{encFx} \leftarrow \text{FRSEIT}[k][n].encFx; \)
   5. \( \text{SetBits}(\text{buffer},1,\text{encFx},\text{len}); \)
   6. \( \text{len} \leftarrow \text{len} + 1; \)
else
   7. while \( n > F_{k+S} \) do
       8. \( k \leftarrow k + S; \)
       9. \( x \leftarrow n \gg F_k; \)
      10. \( \text{LFx} \leftarrow \text{FRSEIT}[k][x].LFx; \)
      11. \( \text{encFx} \leftarrow \text{FRSEIT}[k][x].encFx; \)
      12. \( \text{SetBits}(\text{buffer},1,\text{encFx},\text{LFx}); \)
      13. \( \text{len} \leftarrow \text{LFx} + k + 1; \)
      14. \( n \leftarrow n - \text{FRSEIT}[k][x].Nmin; \)
      15. while \( k > 0 \) do
          16. \( k \leftarrow k - S; \)
          17. if \( n \geq F_k \) then
              18. \( x \leftarrow n \gg F_k; \)
              19. \( n \leftarrow n - \text{FRSEIT}[k][x].Nmin; \)
              20. \( \text{encFx} \leftarrow \text{FRSEIT}[k][x].encFx; \)
              21. \( \text{SetBits}(\text{buffer},\text{len} - k - S,\text{encFx},S); \)
          else
              22. \( \text{SetBits}(\text{buffer},\text{len} - k - S,0,S); \)
          end
      end
5. \( \text{SetBit}(\text{buffer},0,1); \)
```

**Algorithm 5:** A fast encoding algorithm for the Fibonacci code of order 2

A number \( n \) is encoded in parts and individual segments of the bit-length \( S \) are written in \( \text{buffer} \). Due to the fact that the bits of the Fibonacci representation are stored in the reverse order, we need to write segments and bits of the segments in the reverse order as well. When
the bit-length of the Fibonacci representation is not divided by $S$ without remainders, the bit-length of the highest segment is shorter than $S$.

For larger numbers, we first need to find the maximal $k$ in Line 8; this value is the parameter of the Fibonacci right shift. The $k$ value also determines the number of segments used. The input number is encoded by $kS + 1$ segments in $kS$ steps. Since the bit-length of the highest segment can be shorter than $S$, this segment is separately encoded in Lines 9–14. The order $x$ of the current $FRSEIT$’s row is computed by the $k$-Fibonacci right shift in Line 9. The efficient computation of the $k$-Fibonacci right shift by means of Algorithm 1 is utilized. In the $x$-th row of $FRSEIT$, we read the bit-length of the Fibonacci representation included in the current segment in Lines 10 and the encoded Fibonacci representation in Line 11. The bits of the codeword are written in $buffer$\footnote{The reverse operation is a part of the $SetBits$ function, another way is to compute and include the reverse bits in $FRSEIT$ by the $reverse(\text{enc}(F(x)))$ function.} from the position 1 (Line 11); the position 0 is reserved for the 1-bit delimiter. The bit-length of the codeword is computed in Line 13. In Line 14, we subtract the lower bound $n_{\text{min}}$ from $n$. In this way, we remove the bits of the highest segment from $n$. In Lines 15–25, all other segments are similarly encoded and written in $buffer$. Finally, in Line 27, the 1-bit delimiter is written in $buffer$.

**Example 5** (Encoding the number 17,327 with the fast Fibonacci of order 2 algorithm)
Let us have encoding the number 17,327 using the 8-bit segment ($S = 8$). The count of segments needed is computed in Line 8 of Algorithm 5. Since $17,327 \in [F_{16}; F_{24}] = \{2, 582; 121, 393\}$, we find the parameter $k$ of the Fibonacci right shift for the highest segment in Line 8; $k = 16$. This implies the number of segments needed for encoding: $\frac{k}{S} + 1 = \frac{16}{8} + 1 = 3$.

Encoding the highest segment (Segment 2) is depicted in Figure 2. After the 16-th Fibonacci right shift of 17,327, we obtain the order $x$ of the current FRSEIT’s row by $x = 17,327 >> F_{16} = 7$ (Line 9). The result of the shift is depicted in STEP 1 of the figure. In STEP 2, this result is encoded using the 7-th row of FRSEIT by $\text{encF}x = \text{enc}(F(7)) = 1010_2 = 10_{10}$ with the bit-length $\text{LF}x = L(F(7)) = 4$ in Lines 10–11. These bits are reversely written in buffer in Line 12 from the position 1 (see STEP 3 and STEP 4 of this figure); bits of the codeword not used in this computation are grey highlighted. The bit-length of the codeword is computed by $\text{len} = \text{LF}x + k + 1 = 4 + 16 + 1 = 21$ (Line 13). In FRSEIT, we also read the number $17,327 \in [15, 127; 17, 710]$; therefore, the value of other two segments (Segments 0 and 1) is computed by $17,327 - n_{\text{min}} = 17,327 - 15,127 = 2,200$ (Line 14).

Encoding the next segments is carried out in the loop in Lines 15–25. In Line 16, we compute the $k$ parameter of the Fibonacci right shift for Segment 1 by $k = k - S = 16 - 8 = 8$. Encoding Segment 1 is depicted in Figure 3. After the 8-th Fibonacci right shift of 2,200 (the value of Segments 1 and 0) in Line 18, we obtain the order $x$ of the current FRSEIT’s row by $x = 2200 >> F_{8} = 46$. The result of the shift is depicted in STEP 1 of the figure. Consequently, the encoded Fibonacci representation of $x$ is retrieved in the 46-th row of FRSEIT; $\text{encF}x = \text{enc}(F(46)) = 10010101_2 = 149_{10}$ (Line 20). This situation is
depicted in STEP 2 of the figure. In Line 21, we reversely write \( S \) bits of \( \text{encF}x \) in buffer from the position \( \text{len} - k - S = 21 - 8 - 8 = 5 \) (see STEP 3 and STEP 4 of the figure); bits of the codeword not used in this computation are grey highlighted. In FRSEIT, we read that the number \( 2, 200 \in [2, 173; 2, 206] \); therefore, the value of the last segment (Segments 0) is computed by \( 2, 200 - n_{\text{min}} = 2, 200 - 2, 173 = 27 \) (Line 19).

The last Segment (Segment 0) with the value 27 is directly encoded by the 27-th row of FRSEIT: \( \text{encF}x = \text{enc}(F(27)) = 1001001 \) (Line 20). Since \( k = 0 \), the \( k \)-th Fibonacci right shift is not applied in Line 18. In Line 21, we reversely write \( S \) bits of \( \text{encF}x \) in buffer from the position \( \text{len} - k - S = 21 - 0 - 8 = 13 \). The loop ends since there is no other segments to encode. Finally, the 1-bit delimiter is written in buffer in the position 0 (Line 27).

### 6.3 Fast Encoding of Fibonacci Code of Order 3

A fast encoding algorithm for the Fibonacci code of order 3 is shown in Algorithm 6. Let us remember that each codeword of the Fibonacci code of order 3 includes the reverse Fibonacci representation of \( Q \) and the 0111 delimiter (see Section 2.2.2 for more details). The fast encoding algorithm of the Fibonacci code of order 3 is similar to the fast encoding algorithm of the Fibonacci code of order 2; however, it differs in some parts. Special cases, numbers 1 and 2, are encoded in Lines 2–8. In Line 9, we need to compute the bit-length of \( n \). The index \( g \) of the nearest lower Fibonacci sum is computed in Lines 10–11. The efficient computation of the nearest lower Fibonacci sum by means of Algorithm 2 is utilized. The nearest lower Fibonacci sum is utilized to compute the \( Q \) value in Line 12. In Line 13, we compute the bit-length of the codeword.

In Lines 14–15, we compute the maximal parameter \( k \) of the Fibonacci right shift. The \( k \) value also determines the number of segments used by the encoding algorithm. The input number is encoded by \( \frac{k}{3} + 1 \) segments in \( \frac{k}{3} + 1 \) steps. The Fibonacci representation of \( Q \) is appended with 0-bits in Line 21 to extend the bit-length of the codeword to \( m + g \) bits. The number of the 0-bits is computed in Line 20. The 0-bits are written from the position 4 since positions 0–3 are reserved for the 0111 delimiter.

The highest segment with the \( \frac{k}{3} \) order is encoded in Lines 16–18. The order \( x \) of the current FRSEIT’s row is computed by the \( k \)-Fibonacci right shift in Line 16. The efficient computation of the \( k \)-Fibonacci right shift by means of Algorithm 1 is utilized. In the \( x \)-th row of FRSEIT, we read the bit-length of the Fibonacci representation in Lines 17 and the encoded Fibonacci representation in Line 18. The bits of the codeword are written in buffer in Line 19.

Encoding other segments in Lines 22–33 is similar to the encoding of the Fibonacci code of order 2 (see Lines 14–24 of Algorithm 5); however, we use the \( Q \) variable instead of the \( n \) variable. The 0111 delimiter is written in Line 34.

**Example 6** (Encoding the number \( n = 779 \) with the fast encoding algorithm for Fibonacci of order 3)

Let us consider \( n = 779 = 1100001011_2 \). We compute its bit-length \( L(779) = 10 \) (see Line 9 of Algorithm 6). We estimate the \( g \) value of the Fibonacci sum interval \( (S_{g-2}; S_{g-1}) \) as \( g = \text{EstimateFibSum}[10] = 10 \) in Line 10; \( (S_{g-2}; S_{g-1}) = (S_{10-2}; S_{10-1}) = (S_8; S_9) = (326; 600) \). Since \( 10 \notin (326; 600) \), we increment \( g = 10 + 1 = 11 \) (Line 11). Since \( 779 \in (S_{g-2}; S_{g-1}) = (S_{11-2}; S_{11-1}) = (S_9; S_{10}) = (600; 1, 104) \), we can continue with Line 12 where we compute
input: n, a positive integer
output: buffer, a Fibonacci of order 3 codeword of n with the bit-length len

1 $k \leftarrow 0$
2 if $n = 1$ then
3     SetBits(buffer,0,111_2;3);
4     len $\leftarrow 3$
5 else if $n = 2$ then
6     SetBits(buffer,0,0111_2;4);
7     len $\leftarrow 4$
8 else
9     len $\leftarrow L(n)$;
10    $g \leftarrow EstimateFibSum[len]$;
11    while $S_{g-1} < n$ do $g \leftarrow g + 1$;
12    $Q \leftarrow n - S_{g-2} - 1$;
13    len $\leftarrow g + 3$;
14    $k \leftarrow 0$;
15    while $F^{3}_{g+k} < Q$ do $k \leftarrow k + S$;
16    $x \leftarrow Q >> F^{3}_k$;
17    $LFx \leftarrow FRSEIT[k][x].LFx$;
18    encFx $\leftarrow FRSEIT[k][x].encFx$;
19    SetBits(buffer,len-k-LFx,encFx,LFx);
20    $z \leftarrow len - k - LFx - 4$;
21    SetBits(buffer,4,0,z);
22    $Q \leftarrow Q - FRSEIT[k][x].Nmin$;
23    while $k > 0$ do
24        $k \leftarrow k - S$;
25        if $F^{3}_k \le Q$ then
26            $x \leftarrow Q >> F^{3}_k$;
27            $Q \leftarrow Q - FRSEIT[k][x].Nmin$;
28            encFx $\leftarrow FRSEIT[k][x].encFx$;
29            SetBits(buffer,len-k-S,encFx,S);
30        else
31            SetBits(buffer,len-k-S,0,S);
32        end
33    end
34    SetBits(buffer,0,0111_2;4);
35 end

Algorithm 6: A fast encoding algorithm for the Fibonacci code of order 3

$Q = n - S_{g-2} - 1 = n - S_9 - 1 = 779 - 600 - 1 = 178$. The bit-length of the codeword is computed by $len = g + 3 = 11 + 3 = 14$ (Line 13). Since $178 \in [F^{3}_8; F^{3}_{16}] = [149; 19, 513]$, we find the parameter $k$ of the Fibonacci right shift in Lines 14–15; $k = 8$. This implies that the number of segments needed for encoding is $\frac{k}{8} + 1 = \frac{8}{8} + 1 = 2$.

The highest segment (Segment 1) is encoded in Lines 16–18. The Fibonacci right shift is used to obtain the order $x$ of the current FRSEIT’s row by $x = Q >> F^{3}_8 = 100100101_2 >> 8 = 1_F = 1$ (Line 16). In the 1-st row of FRSEIT, we read the encoded Fibonacci representation and the bit-length of $F(x)$; $encFx = enc(F(1)) = 1_2$ and the bit-length $LFx = L(F(1)) = 1$ (Lines 10–11). This bit is written in buffer in the position $len - k - LFx = 14 - 8 - 1 = 5$: buffer = XXXXXXXX1XXXX (Line 20). The 0-bits are appended to make the codeword
bit-length \( m + g \) (Line 20–21). We write \( z = \text{len} - k - LFx - 4 = 14 - 8 - 1 - 4 = 1 \) 0-bits in buffer in the position 4: buffer = XXXXXXXXXXXXXX.

The value included in Segment 0 is computed by \( Q = Q - n_{\text{min}} = 178 - 149 = 29 \) in Line 22. Since the 0-th Fibonacci right shift would be computed in Line 24 for Segment 0, we do not compute any shift, and the Fibonacci representation \( F(30) = 00100101 \) is reversely written from the position \( \text{len} - k - S = 14 - 0 - 8 = 6 \) in buffer in Line 29: buffer = 101001010XXX. Finally, in Line 34, the 0111 delimiter is written in buffer in positions 0 – 3: buffer = 1010010100111.

7 Fast Decoding Algorithms

The fast decoding algorithms process data segment by segment instead of bit by bit as the conventional decoding algorithms. These algorithms are based on a finite automaton with precomputed mapping tables. In the Ph.D. Thesis a general fast decoding algorithm based on an automaton is described as well as the identification of the automaton states. Since the number of automaton states is rather high, we propose two types of the automaton reduction in the Ph.D. Thesis: the first reduction using a similarity of automaton states and the second reduction using a shift operation. In this section, all fast decoding algorithms are described. The fast decoding algorithm for the Elias-delta and Elias-Fibonacci codes is put forward in Section 7.1. In Section 7.2, the fast decoding algorithm for the Fibonacci codes of order 2 and 3 is introduced.

7.1 Fast Elias-delta and Elias-Fibonacci Decoding Algorithms

The fast decoding algorithms for the Elias-delta and Elias-Fibonacci codes are identical; however, the mapping tables are different since the conventional bit-oriented algorithms as well as automatons are different. This algorithm is shown in Algorithm 7.

In Lines 1–3, variables for the state, the current decoded number \( n \), and the bit-length \( \text{len} \) of the current decoded number are initialized. If state < 0, it indicates an error and the algorithm ends in Line 5. In Lines 6–7, the next segment is read from the input stream and it is decoded by means of the mapping table MAP. The new state is set in Line 8. Lines 10–16 cover a situation when the current decoded number is not finished in the actual segment. In other words, the shift operation is used if the number of remaining bits \( \text{len} \geq S \). In Line 10, we compute the number of the remaining bits in the next segment \( \text{len} = \text{len} - S \) which also defines the shift operation in Line 11. In Line 12, we decide whether there are some remaining bits of the codeword in next segment(s) after the shift operation.

After the shift operation, we output the number \( n \) in the result array in Lines 13–14. In Line 16, we set the correct automaton state after the shift operation. Lines 18–25 cover the situation when previously decoded bits are finished in the actual segment. In Lines 18–21, output decoded numbers are written in the result; in one segment, the algorithm can output more then one decoded number. Since the first decoded number can complete bits decoded in the previous segment by the shift operation in Lines 10–16, it is separately processed in Lines 19–20.
input : stream including codewords of the Elias-delta or Elias-Fibonacci code, MAP – Elias-delta or Elias-Fibonacci mapping table

output: result – an array of decoded numbers

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26

while NOT Eos(stream) do
  if state < 0 then break;
  segment ← GetBits(stream,S);
  map ← MAP[state,segment];
  state ← map.NewState;
  if len ≥ S then
    len ← len − S;
    n ← n + (segment << len);
    if len = 0 then
      result ← result ∪ n;
      n ← 0;
    end
  end
  if len < S then state ← len;
  else if map.OutputCount > 0 then
    n ← map.Numbers[0] + n;
    result ← result ∪ n;
    result ← result ∪ \bigcup_{i=1}^{map.OutputCount-1} map.Numbers[i];
  end
  len ← map.Rest;
  n ← map.U << len;
end

Algorithm 7: The fast Elias-delta and Elias-Fibonacci decoding algorithm

The rest of the numbers is written in the result array in Line 21. In Lines 23, len is initialized by the Rest attribute of the mapping table. It means that the remaining bits of the currently decoded number n are stored in the next segment(s). In Line 24, n is initialized with U and it is shifted by the number of the remaining bits len.

Example 7 (The fast Elias-delta decoding algorithm)
Let us consider an example of fast Elias-delta decoding in Figure 4. Rows of MAP accessed in this example are shown in Table 4.

Table 4: Some rows of the mapping table for the fast Elias-delta decoding algorithm (m: and S = 8)

<table>
<thead>
<tr>
<th>State</th>
<th>Segment</th>
<th>NewState</th>
<th>Rest</th>
<th>U</th>
<th>OutputCount</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>104</td>
<td>23</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>{5}</td>
</tr>
<tr>
<td>3</td>
<td>139</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>{4,3,1}</td>
</tr>
<tr>
<td>23</td>
<td>128</td>
<td>3</td>
<td>3</td>
<td>16</td>
<td>0</td>
<td>{ }</td>
</tr>
</tbody>
</table>
After the first segment is read, we access the mapping table for state $= 0$ and segment $= 104$ in Lines 6–7 of the algorithm. The new state of the automaton is set to 23 in Line 8. This state is one of the zero states, it represents 3 zeros of the Elias-delta prefix. These bits belong to the codeword in next segment(s). Since $len = 0$, in Lines 18–22, we obtain the number 5 directly from the mapping table (see the Numbers attribute of MAP) and it is stored in the result array. In Lines 23–24, we initialize $len = 0$ and $n = 0$. Now, the second segment with the value 128 is read. In the mapping table, we access a row of MAP for state $= 23$ and segment $= 128$. The state 3 is set in Line 8. This state is one of the shift states, it indicates that there are 3 bits in the next segment to complete the codeword (see the Rest attribute of MAP). There are not any output numbers for this segment. We set $len = 3$ and $n = 16 << 3 = 128$ in Lines 24–25. The next segment with the value 139 starts with the remaining 3 bits; the sequence 100 is decoded. We obtain the number 4 (see the Numbers attribute of MAP) which is added to $n$: $n = 128 + 4 = 132$ in Line 19. This value is stored in the result array with other two numbers 3 and 1 of Numbers in Lines 20–21.

Let us consider another example of fast Elias-delta decoding in Figure 5. Rows of MAP accessed in this example are shown in Table 5.
Table 5: Some rows of the mapping table for the fast Elias-delta decoding algorithm and \( S = 8 \)

<table>
<thead>
<tr>
<th>State</th>
<th>Segment</th>
<th>NewState</th>
<th>Rest</th>
<th>( U )</th>
<th>OutputCount</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25</td>
<td>8</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>{}</td>
</tr>
<tr>
<td>2</td>
<td>133</td>
<td>37</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>{2}</td>
</tr>
<tr>
<td>8</td>
<td>115</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>{}</td>
</tr>
</tbody>
</table>

in Lines 6–7 of Algorithm 7. The new state of the automaton is set to 8 in Line 8. This state means that the shift operation is applied during the decoding of the next segment. In Lines 23–24, we initialize \( \text{len} = 10 \) and \( n = 3 << 10 = 3,072 \) (see the Rest and \( U \) attributes of MAP).

Now, the second segment with the value 115 is read. We apply the shift operation in Lines 10–16 since \( \text{len} \geq S \) in Line 9. We compute \( \text{len} = 10 - 8 = 2 \) in Line 10 and \( n = 3,072 + (115 << 2) = 3,532 \) in Line 11. The next state is set to 2 in Line 16. The next segment with the value 133 starts with the remaining 2 bits; the sequence 102 is decoded. We obtain the number 2 (see the Numbers attribute of MAP) which is added to \( n \), \( n = 3,532 + 2 = 3,534 \), in Line 19. This value is stored in the result array and the new state is set to 37 in Line 8. This state is one of the bit-length states since the sequence 000101 is a partially decoded bit-length (see Ph.D. Thesis for more details).

Example 8 (The fast Elias-Fibonacci decoding algorithm)

Let us consider an example of fast Elias-Fibonacci decoding in Figure 6. Rows of the mapping table accessed in this example are shown in Table 6.

<table>
<thead>
<tr>
<th>State</th>
<th>Segment</th>
<th>NewState</th>
<th>Rest</th>
<th>OutputCount</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>58</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>{6}</td>
</tr>
<tr>
<td>4</td>
<td>87</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>{5,3}</td>
</tr>
<tr>
<td>11</td>
<td>59</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>{}</td>
</tr>
</tbody>
</table>

After the first segment is read, we access a row for segment = 58 and state = 0 of the mapping table in Lines 6–7. The next automaton state is set to 11 in Line 8. This state
is one of the Elias-Fibonacci prefix states, it represents the partially decoded prefix $10_2$. This sequence is a part of the reverse Fibonacci representation of $L(437)$; $\text{reverse}(F(L(437))) = 9$. In the mapping table, we directly read the number 6 and it is stored in result in Line 20 of the algorithm. We set $n = 0$ and $\text{len} = 0$ in Lines 23–24. Now, the second segment with the value 59 is read and we access a row for segment = 59 and state = 11 of the mapping table in Lines 6–7. We set the new state to 4 in Line 8. This state is one of the shift states, it indicates that there are 4 bits in the next segment to complete the codeword. We set $\text{len} = 4$ and compute the incomplete number $n = 27 << 4 = 432$ in Lines 23–24. The next segment starts with the remaining 4 bits; the sequence $0101_2$ is decoded. In this state, we obtain the numbers 5 and 3 in the mapping table. The first number is added to $n$; we compute $n = 432 + 5 = 437$ in Line 19. The number 437 is stored in result in Lines 20–21 together with the second number 3.

### 7.2 Fast Fibonacci Decoding Algorithm

The fast Fibonacci decoding algorithm is the same for any order $m$ of the Fibonacci code. This algorithm is shown in Algorithm 8, it differs for various orders in the following parts:

- **Mapping table**: The mapping table is based on a different bit-oriented algorithm.
- **Fibonacci left shift operation**: The Fibonacci left shift operations are different for different orders (see Section 3).
- **Final adjustment**: $S_n + 1$ is added in the case of Fibonacci codes of order $m > 2$.

In Lines 1–3, the variables for state, the current output number $n$, and its bit-length $\text{len}$ are initialized. If state $< 0$, it indicates an error; therefore, the algorithm ends in Line 5. In Lines 6–7, the next segment is read from the input stream and it is decoded by means of the mapping table $MAP$. The new state is set in Line 8. If the codeword is finished, it is output in Lines 9–22. If the codeword’s bit-length $\text{len} > 0$, we need to use the Fibonacci left shift; it is applied in Line 11. In Lines 15–18, the Fibonacci sum is added to the decoded number for orders $m > 2$. To select the correct Fibonacci sum, in Line 16, the $k$ value is computed using the remaining bits of the codeword in the actual segment. We also subtract the $m$ value in this line since the codeword’s bit-length must be $m + k$. The result decoded number is add in the result array in Line 19. The $n$ and $\text{len}$ variables are set in Lines 20–21. The Fibonacci left shift is carried out in Line 24. In Line 25, the bit-length $\text{len}$ of the Fibonacci representation of the output number is increased by the bit-length $\text{map.}L_U$ of the decoded part.

**Example 9** (The fast decoding algorithm for the Fibonacci code of order 2)

Let us consider codewords in Figure 7. The rows of the mapping table accessed in this example are shown in Table 7.

After we read the first segment, we access a row for state $= 0$ and segment $= 181$ of the mapping table in Lines 6–7 of Algorithm 8. The new state of the automaton is set to 1 in Line 8. It represents reading the 1-bit at the end of the segment. We directly write the number 4 in the result array in Line 19. We do not apply any shift in Lines 10–14 since $\text{len} = 0$. The incomplete number $U = 7$ is stored in the $n$ variable holding each incomplete
### Algorithm 8: The fast Fibonacci decoding algorithm for Fibonacci codes of order $m$

| input : stream including codewords of the Fibonacci code of order $m$ |
| output: result – array of decoded numbers |

```plaintext
1. $n \leftarrow 0$
2. $state \leftarrow 0$
3. $len \leftarrow 0$
4. while NOT Eos(stream) do
   5. if state < 0 then break;
   6. segment ← GetBits(stream, S);
   7. map ← MAP[state, segment];
   8. state ← map.NewState;
   9. for $i \leftarrow 0$ to map.OutputCount − 1 do
      10. if len > 0 then
          11. $n \leftarrow n + (map.Numbers[i] \ll F \cdot len)$;
      else
          12. $n \leftarrow map.Numbers[i]$;
      end
      13. if $m > 2$ then
          14. $k \leftarrow len + map.L - m$;
          15. $n \leftarrow n + S_{k - 2} + 1$;
      end
      16. result ← result ∪ $n$;
      17. $n \leftarrow 0$;
      18. $len \leftarrow 0$;
   20. end
   21. if map.LU > 0 then
      22. $n \leftarrow n + (map.U \ll F \cdot len)$;
      23. $len \leftarrow len + map.LU$;
   25. end
   26. end
   27. end
```

Example 10 (The fast decoding algorithm for the Fibonacci code of order 3)

Let us consider codewords in Figure 8. The rows of the mapping table accessed in this example are shown in Table 8.

After we read the first segment, we access a row for state = 0 and segment = 139 of the mapping table in Lines 6–7 of the algorithm. In Line 24, the incomplete number $U = 14$ is
Figure 7: An example of the fast Fibonacci of order 2 decoding algorithm for $S = 8$

Table 7: Some rows of the mapping table for the fast Fibonacci of order 2 decoding algorithm and $S = 8$

<table>
<thead>
<tr>
<th>State</th>
<th>Segment</th>
<th>NewState</th>
<th>$U$</th>
<th>$L_U$</th>
<th>OutputCount</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>181</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>{4}</td>
</tr>
<tr>
<td>1</td>
<td>165</td>
<td>1</td>
<td>31</td>
<td>7</td>
<td>1</td>
<td>{0}</td>
</tr>
<tr>
<td>1</td>
<td>114</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>{2}</td>
</tr>
</tbody>
</table>

Figure 8: An example of the fast Fibonacci of order 3 decoding algorithm for $S = 8$

stored in the $n$ variable holding each incomplete number from the previous segment. Its bit-length is set to $len = 6$. The new state of the automaton is set to 2 in Line 8. It represents reading the sequence $11_2$ at the end of the segment. We read the second segment and we access a row for $state = 2$ and $segment = 28$ of the mapping table in Lines 6–7. Since $len > 0$, we use the Fibonacci left shift in Line 11: $n = 14 + 3 \ll F6 = 14 + 125 = 139$. In Line 16, we compute $k = 6 + 8 - 3 = 11$. In Line 17, we compute the output number: $n = n + S_{k-2} + 1 = 139 + S_9 + 1 = 139 + 600 + 1 = 740$. The number 740 is stored in the $result$ array in Line 19. We set $n = U = 0$ and $len = L_U = 2$ in Lines 24–25. The $n$ value is completed in the next segment(s).

Table 8: Some rows of the mapping table for the fast Fibonacci of order 3 decoding algorithm and $S = 8$

<table>
<thead>
<tr>
<th>State</th>
<th>Segment</th>
<th>NewState</th>
<th>$U$</th>
<th>$L_U$</th>
<th>OutputCount</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>139</td>
<td>2</td>
<td>14</td>
<td>6</td>
<td>0</td>
<td>{}</td>
</tr>
<tr>
<td>2</td>
<td>28</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>{3}</td>
</tr>
</tbody>
</table>
8 Experimental Results

In our experiments, we tested all proposed fast encoding and decoding algorithms and compared these algorithms with conventional bit-oriented algorithms\(^3\). The experiments were executed on a PC with Intel Xeon 2.93 GHz, 32 GB RAM; Windows Server 2008 x64. In this section, we use abbreviations ED, EF, F2, and F3 for Elias-delta, Elias-Fibonacci, Fibonacci of order 2, and Fibonacci of order 3, respectively.

![Figure 9: The uniform distribution, normal distribution for \( \sigma^2 = 100 \), and exponential distribution for \( \lambda^2 = 100 \)](image)

The test collections used in the experiments included 10,000,000 randomly generated numbers; we utilized uniform, normal, and exponential distribution functions for generating the random numbers (see Figure 9).

Tests with real data collections are not included in this paper, they are included in Pd.D. Thesis where we tested XML Node Streams and Text Compression.

Tested collections included in this paper were as follows:

- 8, 16, 24, 32-bit – collections of uniformly distributed random numbers ranging from 1 to 255, 256 to 65,535, 65,536 to 16,777,215, and 16,777,216 to 4,294,967,295, respectively.

- Uniform – a collection of uniformly distributed random numbers ranging from 1 to 4,294,967,295

- Normal – a collection of normally distributed random numbers ranging from 1 to 4,294,967,295; we used the normal distribution function \( f(x) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{x^2}{2\sigma^2}}, \) where \( \sigma^2 = 2^{32} \).

- Exponential – a collection of exponentially distributed random numbers ranging from 1 to 4,294,967,295; we used the exponential distribution function \( f(x) = \frac{1}{\lambda}e^{-\frac{x}{\lambda}}, \) where \( \lambda^2 = 2^{32} \).

\(^3\)The fast coding library can be downloaded from [http://db.cs.vsb.cz/SubPages/Projects/Projects.aspx](http://db.cs.vsb.cz/SubPages/Projects/Projects.aspx)
8. Experimental Results

8.1 Compression Ratio

Although studying the compression ratio of variable-length codes is beyond the scope of this paper, in this section, we show the compression ratio, since it is one of the parameters of each code (together with encoding and decoding times). As a baseline, we selected the fixed-length standard binary representation. The size of encoded data for Normal, Exponential, and Uniform 32-bit collections is shown in Table 9 and Figure 10.

Table 9: The size of encoded data for test collections [MB]

<table>
<thead>
<tr>
<th>Test Collection</th>
<th>Standard binary</th>
<th>Elias-delta</th>
<th>Fibonacci 2</th>
<th>Fibonacci 3</th>
<th>Elias-Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td>38.15</td>
<td>47.69</td>
<td>53.88</td>
<td>44.99</td>
<td>45.31</td>
</tr>
<tr>
<td>Exponential</td>
<td>38.15</td>
<td>27.27</td>
<td>27.44</td>
<td>24.15</td>
<td>25.76</td>
</tr>
<tr>
<td>Normal</td>
<td>38.15</td>
<td>27.20</td>
<td>27.30</td>
<td>24.03</td>
<td>25.67</td>
</tr>
</tbody>
</table>

We see that the variable-length codes provide the higher compression ratio for all test collections with an exception of the Uniform collection; however, the encoding of a collection with the uniform distribution is the worst case scenario for each compression method. Evidently, the Fibonacci code of order 3 provides the best compression ratio; however, results of the Elias-Fibonacci code are very similar.

![Size of Encoded Data](image)

Figure 10: The size of encoded data for test collections

8.2 Encoding Algorithms

In Table 10 and Figure 11, the encoding times of all conventional and fast encoding algorithms are shown. We see that the Elias-delta and Elias-Fibonacci conventional encoding algorithms outperform both Fibonacci conventional encoding algorithms. Fast algorithms of both Fibonacci codes achieved faster encoding times for the 8-bit test collection than other fast algorithms (and the fast algorithm of Fibonacci of order 3 outperforms the fast algorithm of Fibonacci of order 2). On the other hand, when we consider a test collection with the larger domain, fast encoding of the Elias-Fibonacci code is shown to be the most efficient. Whereas the


Table 10: Encoding time [ms] and Speed-up ratio [times]

<table>
<thead>
<tr>
<th>Test Collection</th>
<th>Conventional</th>
<th>Fast</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ED</td>
<td>F2</td>
</tr>
<tr>
<td>8-bit</td>
<td>605</td>
<td>1,305</td>
</tr>
<tr>
<td>16-bit</td>
<td>962</td>
<td>2,198</td>
</tr>
<tr>
<td>24-bit</td>
<td></td>
<td>1,217</td>
</tr>
<tr>
<td>32-bit</td>
<td></td>
<td>1,598</td>
</tr>
<tr>
<td>Uniform</td>
<td></td>
<td>1,629</td>
</tr>
<tr>
<td>Exponential</td>
<td></td>
<td>1,058</td>
</tr>
<tr>
<td>Normal</td>
<td></td>
<td>1,037</td>
</tr>
<tr>
<td>Avg.</td>
<td></td>
<td>1,158</td>
</tr>
<tr>
<td>Speed-up ratio</td>
<td>7.1×</td>
<td>12.1×</td>
</tr>
</tbody>
</table>

The processing time of the fast Elias-delta encoding algorithm is close to the fast Elias-Fibonacci algorithm, the fast encoding algorithm of the Fibonacci code of order 3 is less efficient, the encoding time of the fast algorithm for the Fibonacci code of order 2 is between Elias-delta and Fibonacci of order 3.

![Encoding Time [ms]](image)

Figure 11: Encoding times for conventional and fast algorithms

The average speed-up ratio for all test collections is shown in the last line of Table 10. All fast encoding algorithms achieved faster encoding times than the conventional algorithms, the speed-up ratio ranges from 7.0× for the Fibonacci code of order 3 to 12.5× for the Elias-Fibonacci code. The speed-up ratios for the Elias-Fibonacci and Fibonacci of order 2 codes are approximately 2× higher than speed-up ratios for the Elias-delta and Fibonacci of order 3 codes.

### 8.3 Decoding Algorithms

In Table 11 and Figure 12, the decoding times of all conventional and fast algorithms are shown. Although the most efficient compression ratio is achieved by the Fibonacci code of order 3 (see Table 9), the fast decoding algorithm of the Elias-Fibonacci code outperforms all other fast decoding algorithms from the processing time point of view. The fast Fibonacci decoding algorithms of both orders utilize the Fibonacci shift operation, which is more time consuming compared to a common shift operation applied in the case of the fast Elias-delta and Elias-Fibonacci decoding algorithms. The Fibonacci code of order 3 provides the less efficient
decoding time than the Fibonacci code of order 2 in all tests, because this algorithm includes more operations (see Section 7.2). Consequently, Fibonacci of order 3 can be used in cases when we prefer the higher compression ratio rather than the lower decoding time. On the other hand, Elias-Fibonacci is used in cases when we prefer the lower decoding time rather than the highest compression ratio.

Table 11: Decoding time [ms] and Speed-up ratio [times] for all conventional and fast algorithms

<table>
<thead>
<tr>
<th>Test Collection</th>
<th>Conventional</th>
<th>Fast</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ED</td>
<td>F2</td>
</tr>
<tr>
<td>8-bit</td>
<td>357</td>
<td>699</td>
</tr>
<tr>
<td>16-bit</td>
<td>589</td>
<td>1,370</td>
</tr>
<tr>
<td>24-bit</td>
<td>718</td>
<td>2,049</td>
</tr>
<tr>
<td>32-bit</td>
<td>937</td>
<td>2,704</td>
</tr>
<tr>
<td>Uniform</td>
<td>935</td>
<td>2,716</td>
</tr>
<tr>
<td>Exponential</td>
<td>626</td>
<td>1,421</td>
</tr>
<tr>
<td>Normal</td>
<td>621</td>
<td>1,418</td>
</tr>
<tr>
<td>Avg.</td>
<td>683</td>
<td>1,768</td>
</tr>
<tr>
<td>Speed-up ratio</td>
<td>3.1×</td>
<td>6.7×</td>
</tr>
</tbody>
</table>

Figure 12: Decoding time for all conventional and fast algorithms

In the case of an 8-bit data collection, fast algorithms do not utilize shift operations; therefore, the decoding times are very similar. In this case, Fibonacci codes of any order can be the most efficient choice because these codes provide higher compression ratios than the Elias-delta or Elias-Fibonacci codes.

In the last row of Table 11, we see the average speed-up ratios of all fast decoding algorithms for all collections. The speed-up ratio ranges from 3.1 of the Elias-delta code to 8.9 of the Fibonacci of order 3 code.

9 Conclusion

This paper begins with an overview of the most used variable-length codes; Huffman, Golomb, Elias, and Fibonacci of order 2 and 3 codes are described in more detail. Moreover, we introduced a new code titled the Elias-Fibonacci code.
Next sections described the fast encoding and decoding algorithms for the Elias-delta, Fibonacci of order 2 and 3, and Elias-Fibonacci codes. The fast algorithms are based on a strong theoretical background, e.g. Fibonacci shift operations and their efficient computation and the efficient computation of the nearest lower Fibonacci sum are introduced. As result, the fast encoding algorithms are up-to $12.5\times$ faster than the conventional encoding algorithms and the fast decoding algorithms are up-to $8.9\times$ faster than the conventional decoding algorithms.

The new Elias-Fibonacci code achieved the fastest encoding and decoding time on average for domains of the bit-length $\geq 16$. On the other hand, when the 8-bit domain is considered, the Fibonacci code of order 3 outperforms other codes. Fibonacci of order 3 also achieved the lowest size of encoded data; however, results of Elias-Fibonacci are very similar. Consequently, we provided results showing various situations where a concrete code outperforms each other.

In our future work, we want to develop fast algorithms for other universal codes and we also plan to apply the universal codes in more compression algorithms.

Author’s Bibliography


Author’s Bibliography not related to this work

Recorded: SCOPUS.

Bibliography


